Promote-IT: An efficient Real-Time Tertiary-Storage Scheduler

Maria Eva Lijding, Sape Mullender, Pierre Jansen
Distributed and Embedded Systems Group (DIES), University of Twente
P.O.Box 217, 7500AE Enschede, The Netherlands
{lijding, sape, jansen}@cs.utwente.nl
tel +31-53-4893770, fax +31-53-4984590

Abstract

Promote-IT is an efficient heuristic scheduler that provides QoS guarantees for accessing data from tertiary storage. It can deal with a wide variety of requests and jukebox hardware. It provides short response and confirmation times, and makes good use of the jukebox resources. It separates the *scheduling* and *dispatching* functionality and effectively uses this separation to dispatch tasks earlier than scheduled, provided that the resource constraints are respected and no task misses its deadline.

To prove the efficiency of Promote-IT we implemented alternative schedulers based on different scheduling models and scheduling paradigms. The evaluation shows that Promote-IT performs better than the other heuristic schedulers. Additionally, Promote-IT provides response-times near the optimum in cases where the optimal scheduler can be computed.

1 Introduction

Today multimedia data is generally stored in secondary storage (hard disks) and is from there delivered to the users. However, the amount of storage capacity needed for a multimedia archive is large and constantly growing with the expectations of the users. Tertiary-storage jukeboxes can provide the required storage capacity in an attractive way if the data can be accessed with real-time guarantees.

A *jukebox*[1] is a large tertiary storage device that can access data from a large number of removable storage media (RSM, for example DVDs or tapes) using a small number of drives and one or more robots to move RSM between their shelves and the drives. A central problem with this setup is that the RSM switching times are high, in the order of tens of seconds. Thus, multiplexing between files may be many orders of magnitudes slower than on a hard drive, where it takes only a few milliseconds. The second important problem is the potential for resource contention that results from the shared resources in the jukebox.

[1] We use the term *jukebox* to refer to any type of *Robotic Storage Library (RSL)*

Our *hierarchical multimedia archive (HMA)* is a service that provides flexible real-time access to data stored in tertiary storage. The HMA can serve complex requests for the real-time delivery of any combination of media files it stores. A request consists of a deadline and a set of *request units* for individual files (or part of files). Such requests can for instance result from a database query to compile a historical background for news on-the-fly, or from a personalized entertainment program consisting of music video clips. The HMA can also be used to provide real-time guarantees in the access of scientific data, e.g., earth measurements, weather forecast. In the latter cases it is especially important to be able to tell the users in advance when the data will be available in secondary storage.

Tertiary storage plays an important role in supercomputing environments and scientific computing. Essential to these environments is the capacity to deal with petabytes of data that must be easily accessible to geographically distributed scientists. The storage hierarchy that stores the data must be transparent to the users, except for the delays of accessing data in tertiary storage. The IEEE Mass Storage System Reference Model [7] describes the characteristics such systems should posses. Multiple *hierarchical storage management (HSM)* systems have been developed, both conforming to the reference model and prior to it. Some examples are the High Performance Storage System (HPSS) of the National Storage Laboratory [23], and the Storage and Archive Manager File System (SAM-FS) of Fujitsu [9]. The openness of the reference model permits to include specific real-time services as future interfaces [23]. However, no HSM so far supports real-time services. Our HMA can be incorporated in the reference model as a *Storage Server* component.

The HMA uses secondary storage as a buffer and cache for the data in its tertiary-storage jukeboxes. The *jukebox scheduler* is the key component of the HMA that guarantees the in-time promotion of data from tertiary storage to secondary storage. Apart from providing real-time guarantees, the scheduler also tries to minimize the number of rejected requests, minimize the response time for ASAP requests, minimize the confirmation time, and optimize hardware utilization.

We use a new design of jukebox schedulers, where the *scheduling* and *dispatching* functionality are clearly separated. This separation allows us to improve the performance of the system, because the optimality criteria of both functions are different. The goal of the *schedule builder* is to find feasible schedules for the requested data. Thus, the scheduler tries to build schedules as flexible as possible and is not concerned about the optimal use of the resources. The *dispatcher*, instead, is concerned about utilizing the jukebox resources in an efficient manner. We introduce the concept of *early dispatching*, by which a dispatcher can dispatch the tasks earlier than scheduled as long as the resource constraints are respected and no task misses its deadline.

The first step to build an efficient scheduler is to understand the scheduling problem thoroughly. On the one hand, we *model the hardware* and identify the parameters that define the hardware behavior. Our model is flexible and can represent any present and expected future jukebox hardware. On the other hand, we *formalize the scheduling problem* using scheduling theory so that its characteristics and complexity can be analyzed, and the problem can be classified and compared with other scheduling problems. Given the complexity

of the scheduling problem we are dealing with, there are many different ways in which it can be modeled.

The most important of these models is the *minimum switching model*, which models the problem as a *flexible flow shop* with three stages—load, read, unload. The model uses shared resources to guarantee mutual exclusion in the use of the jukebox resources. This model puts only a small restriction on the utilization of the resources, which additionally results in better use of the resources and system performance. The model requires that once an RSM is loaded in a drive, all the requested data of the RSM is read before the RSM is unloaded. Thus, the schedules that can be built with this model have a minimum number of switches.

Promote-IT is based on the minimum switching model. For every incoming request it builds a new schedule that includes all the previously scheduled request units plus the request units of the new request. It uses an efficient heuristic algorithm to find a solution to an instance of the minimum switching model on-line. Promote-IT can deal with any type of request and jukebox hardware. Additionally, it provides short response times and confirmation times, and makes good use of the jukebox resources.

We defined different scheduling strategies for Promote-IT, which vary in the way in which the jobs are added to the schedule. These strategies can be classified as *Front-to-Back* (*earliest deadline first (EDF)* and *earliest starting time first (ESTF)*) and *Back-to-Front* (*latest deadline last (LDL)* and *latest starting time last (LSTL)*). When using Front-to-Back, each job is scheduled as early as possible, while with Back-to-Front, each job is scheduled as late as possible. When using Back-to-Front, Promote-IT profits strongly from the separation of scheduling and dispatching. The scheduler creates schedules with idle times that are used by the dispatcher to dispatch tasks early. This combination proves useful in many cases, especially when the use of a shared robot is the bottleneck in the system.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents some more details about Promote-IT. Section 4 evaluates Promote-IT, comparing it's capabilities and performance with that of other schedulers. Finally, Section 5 concludes the paper.

2 Related Work

We first discuss two schedulers that can be used in a HMA. In Section 4 we compare the performance of these schedulers with that of Promote-IT. Later in this section we briefly discuss schedulers for more simple requests, schedulers with unsolved contention problems and schedulers for discrete-media.

Lau et al. [15] present an aperiodic scheduler for Video-on-Demand systems that can use two scheduling strategies: *aggressive* and *conservative*. When using the aggressive strategy each job is scheduled and dispatched as early as possible, while when using the conservative strategy each job is scheduled and dispatched as late as possible. These two strategies

are similar to the EDF and LDL strategies that we use in Promote-IT. An important difference between the strategies of Lau et al. and Promote-IT is that their strategies dispatch the tasks in the same sequence and time as assigned in the schedule. Thus, the conservative strategy performs poorly, because it leaves the resources idle, even when there are tasks that need executing. Another important difference is that their algorithm handles the jobs to include in the schedule as formed by a *read task* and a *switch task*. The switch task is scheduled as a unity, although it involves unloading the RSM loaded in the drive and loading the new RSM. Lau et al. assume that all the drives are identical and that the switching time is constant, independently of the drive and shelf involved. The former assumption is reasonable in many jukeboxes, but makes the algorithm difficult to generalize to the case with non-identical drives. The latter assumption is not reasonable in most of the large jukeboxes and forces the use of worst-case switching times when building the schedules. Using more accurate switching times provides better schedules.

Federighi et al. [8] use requests similar to those of the HMA. In their system the videos may be stored in multiple objects, with different sound tracks and subtitles corresponding to each video. The requests in their system have *soft deadlines*, e.g., the data should be available at around eight o'clock. Federighi et al. are mainly concerned about balancing the load on distributed video file servers, which are placed near the users [2]. An important difference with our approach is that, even if the requests consist of multiple objects, the playback only begins once all the objects are available at the video file servers. We refer to this type of approach as *Fully-Staged-Before-Starting (FSBS)*.

There are multiple proposals for scheduling continuous data stored in one RSM [4, 6, 5, 21, 25, 11]. The main difference among these proposals is whether the data should be fully staged before starting, streamed directly to the user, or pipelined (i.e., the data of a request can be consumed while other data of the request is being staged). We show in our work that pipelining the data is the best approach.

Various authors try to solve the problem of providing access to data contained in multiple RSM, however their schedulers suffer from unsolved contention problems [1, 16, 12, 3]. Therefore, these schedulers cannot guarantee that the real-time deadlines are always met. We analyze the faults in these schedulers in [17].

There are numerous proposals for scheduling requests for discrete data [24, 10, 22, 19, 20]. The goal of these schedulers is to minimize the average response time. In all cases, the conclusion is that as much data as possible should be read from an RSM when the RSM is loaded in the drive. These results support the minimum switching model that we use in Promote-IT.

More et al. [20] are concerned with performing queries on data that is stored in multiple tapes. Thus, a query may have multiple request units without real-time constraints. Their goal is to minimize the response time of each query. They model the scheduling problem as a two-machine flow-shop with additional constraints. In their model, the unload and load of a tape are coupled. They propose the longest transfer-time first (LtF) algorithm that for each query starts reading first the data of the sub-queries that require the longest transfer time. If there are multiple sub-queries for the same tape they use the SORT algorithm proposed by

Hillyer et al. [13] to decide the order in which the sub-queries should be read. The rationale behind the LtF algorithm is that while the data of the longest sub-query is being read, there is time to switch the tapes on the other drives and read the data corresponding to the shorter sub-queries. Through analytical analysis and simulations they show that LtF provides short response times.

In our HMA we can represent the type of requests More et al. are concerned with as a request with multiple request units with the same delta deadline (see Section 3 for details about the request of the HMA). The strategies of Promote-IT that use the latest starting time as parameter to sort the jobs build similar schedules to those of LtF for this type of requests, even if the length of the transfer is not the scheduling parameter used by Promote-IT. Given a set of RSM with the same deadline and different transfer times, the ones with longer transfer times will have earlier latest-starting-times. Thus, these strategies of Promote-IT will also schedule the RSM to begin earlier.

3 Promote-IT

A *request* r_i, which a user issues to the Hierarchical Multimedia Archive, consist of a deadline and a set of l_i request units u_{ij} for individual files (or part of files). The request can represent any kind of static temporal relation between the request units. Formally we express the user request structure in the following way:

$$r_i = (\tilde{d}_i, asap_i, \textit{maxConf}_i, \{u_{i1}, u_{i2}, \ldots, u_{il_i}\})$$
$$u_{ij} = (\Delta\tilde{d}_{ij}, m_{ij}, o_{ij}, s_{ij}, b_{ij})$$

The *deadline* $\tilde{d}_i$ of the request is the time by which the user must have guaranteed access to the data. The flag $asap_i$ indicates if the request should be scheduled as soon as possible. The user may specify no deadline ($\tilde{d}_i = \infty$) if the only restriction is that the request should be scheduled ASAP. The maximum confirmation time $\textit{maxConf}_i$ is the time the user is willing to wait in order to get a confirmation from the system, which indicates if the request was accepted or rejected. The system must provide a confirmation before making the data available, so $\textit{maxConf}_i \leq \tilde{d}_i$.

The relative deadline of the request unit $\Delta\tilde{d}_{ij}$ is the time at which the data of the request unit should be available, relative to the starting time of the request. The other parameters of the request unit m_{ij}, o_{ij}, s_{ij} and b_{ij} represent the RSM where the data is stored, the offset in the RSM, the size of the data, and the bandwidth with which the user wants to access the data, respectively.

The starting time of the request must not be later than its deadline, so $st_k \leq \tilde{d}_k$. If the request is ASAP, the scheduler assigns the request the earliest possible starting time st_k that will allow it to be incorporated into the system. Thus, the scheduler must find the minimum starting time st_k that makes $\mathcal{U}'$ schedulable, where $\mathcal{U}'$ is the set of request units that need scheduling. The scheduler tries different candidate starting times st_k^x and selects the earliest feasible st_k^x. assigns it the starting time corresponding to its deadline. If the

deadline of the request cannot be met, then the scheduler puts the request in the list of unscheduled requests until it can schedule it or *maxConf*$_i$ is reached and the request is rejected.

The structure of the scheduling algorithm of Promote-IT is the following:

1. Generate a candidate starting time st_k^x and update the deadline of each request unit so that $\tilde{d}_{kj} = st_k^x + \Delta\tilde{d}_{kj}$. The algorithm uses a variation of the bisection method for finding roots of mathematical functions.
2. Model $\mathcal{U}'$ as an instance of the minimum switching model. We represent the instance of the problem by the set $\mathcal{J}$ of jobs to schedule.
3. Compute the *medium schedules*. For each RSM, compute m medium schedules—one MS for each drive. Set the parameters of the duration and deadline of the read tasks T_{2j} to the corresponding values of the computed MS
4. Compute the resource assignment. The algorithm must incorporate each job $J_j \in \mathcal{J}$ into the schedule. If the algorithm succeeds in finding a valid resource assignment, the output of this step is a feasible schedule S^x; otherwise $S^x = \emptyset$. The pair (S^x, st_k^x) is incorporated into the list of analyzed solutions.
5. Repeat from step 1 until the bisection stop-criteria is fulfilled for the list of candidates, i.e. the time difference between the last unsuccessful and first successful candidate is smaller than a threshold.
6. Select the best solution. The best solution is the earliest candidate starting time for which step 4 could compute a feasible schedule ($\min\{st_k^x \mid S^x \neq \emptyset\}$). If there is no such st_k^x, the request r_k is placed in the list of unscheduled requests to be scheduled at a later time. Otherwise, the scheduler confirms the starting time st_k to the user and replaces the active schedule with the new feasible schedule.

In the *minimum switching model* we limit the number of jobs to one per RSM. This model requires that all the requested data from an RSM must be read before the RSM is unloaded from a drive. The processing environment of our model is a *flexible flow shop* with three stages (FF_3). The first stage is to load an RSM to a drive, the second stage is to read the data from an RSM and the third is to unload the RSM. The jobs to be processed are of the form $J_j = \{T_{1j}, T_{2j}, T_{3j}\}$, with one task for each stage.

Both the drives and robots may all have different characteristics. Therefore, the processors at each stage are modeled as *unrelated*. In the first stage there are l processors representing the l loader robots. In the second stage there are m processors representing the drives. In the third stage there are u processors representing the unloader robots. The robots in the first and third stage may have some elements in common and in the worst case all the elements will be the same: when all robots are able to load and unload. Because the robots may be limited to serve only a subset of drives and shelves, there are jobs that can be executed only in a subset of resources. In the model we indicate this by using *machine eligibility restrictions*.

The processing time of a reading task T_{2j} is determined by computing a separate schedule for all request units that are grouped into J_j. We call this schedule for an RSM a *Medium Schedule (MS)*. An MS determines in which order the data must be read once the RSM is in the drive. As the drives may be non-identical, we compute a separate MS for each drive. The optimization criterion for an MS is to maximize the time at which the RSM has to be loaded in a drive to start reading data from it, in such a way that the deadlines of the request units are met. In other words we want to determine the latest possible starting time of the read. If the RSM is already loaded in a drive, the goal is to read the requested data before the RSM must be unloaded.

In step 4 we use a branch-and-bound algorithm to prune the tree of possible assignments of jukebox resources to the jobs in $\mathcal{J}$. The branch-and-bound algorithm uses the *best-drive heuristic* to choose which drive will be tried first to schedule a job and prune from the tree the branches corresponding to drives which offer a worse solution. When pruning the tree, the algorithm may be throwing away a feasible solution that an optimal scheduler would find. But searching the whole tree of solutions is computationally unacceptable. For comparison, we have also implemented an optimal scheduler, but it can take up to several days to compute a feasible schedule for one new request, in contrast to the few milliseconds needed by Promote-IT.

The jobs are incorporated to the schedule using any of the four strategies presented in Section 1. None of the strategies is absolutely better than the others, because each strategy can find schedules that cannot be found by the others. However, ESTF performs best in most cases, and when the system load is very high it is convenient to use LDL. The difference in performance between the different strategies is small when compared with other schedulers. Therefore, in the next section we use only ESTF and LDL as representatives of Promote-IT.

4 Evaluation

To prove the efficiency of Promote-IT, we implemented alternative schedulers based on different scheduling models and scheduling paradigms. On the one hand, we designed two new schedulers: the *jukebox early quantum scheduler (JEQS)* and the *optimal scheduler*. On the other hand, we extended some heuristic schedulers proposed in the literature (see Section 2): the *extended aggressive strategy*, the *extended conservative strategy* and *Fully-Staged-Before-Starting (FSBS)*. Our extensions are able to deal with the requests used in the HMA, and with jukeboxes with different drive models and multiple robots. Furthermore, they do not assume constant switching and reading times. The extended schedulers have better properties than the original ones, while still keeping the features of the original schedulers that we consider most important to evaluate.

The *jukebox early quantum scheduler (JEQS)* is a periodic scheduler. The basic heuristic used by a periodic scheduler is to represent the requests as periodic tasks. A restriction of periodic schedulers is that they can be used only for some special use cases of HMA, as Video-on-Demand, because they are unable to deal with complex requests. Addition-

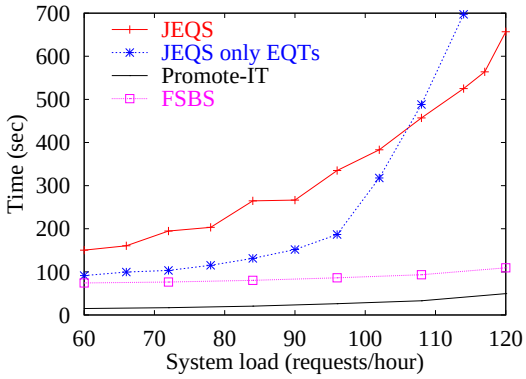
ally, periodic schedulers have serious problems in avoiding resource-contention problems. JEQS solves these problems by using the robots and drives in a cyclic way. The robot exchanges the contents of each drive at regular, fixed intervals. This results in a cyclic use of the drives, which are dedicated to reading data of an RSM while the other drives are being served by the robot. To our best knowledge, JEQS is the only correct periodic jukebox scheduler. The other periodic jukebox schedulers presented in the literature do not deal correctly with the resource-contention problem.

JEQS uses the scheduling theory on *early quantum tasks (EQT)* presented in [14]. An early quantum task is a task whose first instance is executed in the next quantum after its arrival and the rest of the instances are scheduled in a normal periodic way with the release time immediately after the first execution. Although, JEQS is generally able to start incoming requests in the next cycle of a drive, its performance is much worse than that of any of the aperiodic schedulers.

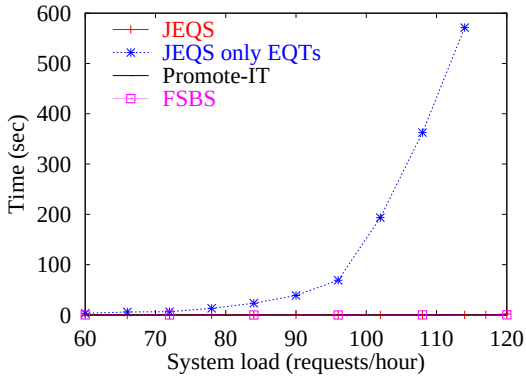
The *optimal scheduler* is a scheduler that computes the minimum response time for each incoming request. The objective of this scheduler is to be used as a baseline for evaluating the quality of the heuristic schedulers. The optimal scheduler cannot be used in a real environment due to its computing-time requirements. The computing time increases exponentially with the complexity of the requests and the system load. Therefore, we can only use it for evaluation of small test sets and relatively low system load. The comparisons that we performed show that the performance of Promote-IT is near the optimum, at least under these special testing conditions.

The simulations shown in this section were performed with JukeTools [18], using in each case representative jukebox architectures and request sets. Except when evaluating the optimal scheduler, the size of the cache is 10% of the jukebox capacity. The number of requests that can be handled by the schedulers in each of the examples shown depend on the request set and the hardware used. In each of the individual comparisons all the schedulers use the same request set and hardware simulation. As we show, some schedulers can handle load better than others, and in general the load shown in the graphics is determined by the load that can be handled by the most restrictive scheduler being compared.

Figure 1(a) compares the response time of aperiodic and periodic schedulers. Aperiodic scheduling is represented by Promote-IT and FSBS, periodic scheduling by JEQS. In this comparison the performance of Promote-IT is representative for the performance of the extended aggressive strategy and extended conservative strategy, because the difference in performance among these schedulers is negligible when compared with the difference among Promote-IT, FSBS and JEQS. For JEQS we consider two variations: scheduling normal quantum tasks (shown in the plots as 'JEQS') and scheduling only EQTs (shown in the plots as 'JEQS only EQTs'). We use FSBS in this comparison, because even though FSBS is very simple in many cases it performs better than JEQS. FSBS has a similar behavior to a First-Come-First-Serve scheduler, which virtually means that no serious scheduling is done. It first serves a request completely and only then it provides access to the data of the request.



(a) Mean Response Time



(b) Mean confirmation time.

Figure 1: Aperiodic vs. periodic scheduling.

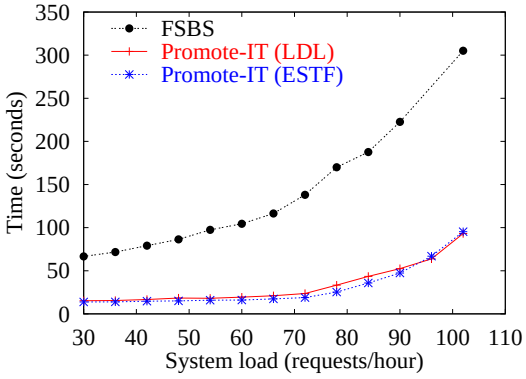


Figure 2: Pipelining vs. full staging. Mean Response Time.

The request set consists of 1000 ASAP requests that follow a Zipf distribution. Each request corresponds to one long-video file, because of the restrictions imposed by JEQS. To be able to use JEQS the request must be only for data stored in one RSM in a contiguous fashion. Additionally, JEQS needs the data to be continuous-media. When using Promote-IT the request is split in request units of 100 MB in size. The requests cannot be rejected, i.e., deadline and maximum confirmation time are infinite. The data in the jukebox is stored in double-layered DVDs and each video is stored completely in one disk. However, one disk may store multiple videos.

The jukebox has four identical DVD drives and one robot. The load time is between 21.8 and 24.8 seconds, while the unload time is between 14.3 and 17.4 seconds. The drives use CAV technology and have a transfer speed that ranges between 7.96 and 20.53 MBps and a maximum access time of 0.17 seconds.

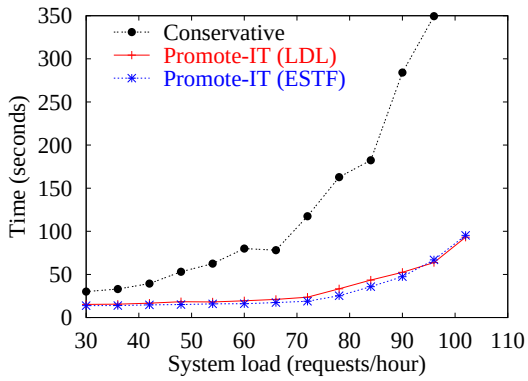
The response time of Promote-IT is much shorter than the response time of JEQS. As the system load increases, the performance of FSBS is also better than that of JEQS. JEQS uses the resources poorly, because it performs multiple switches for reading data from an RSM. In contrast, Promote-IT and FSBS use the resources efficiently by performing the minimum amount of switches required to read the data.

The confirmation time of the aperiodic schedulers is shorter that that of JEQS (see Figure 1(b)). The main difference can be seen with 'JEQS only EQTs', because this scheduler waits to accept a request until it can schedule it as an EQT. As the system load increases, the possibilities of accepting a request as an EQT diminish drastically.

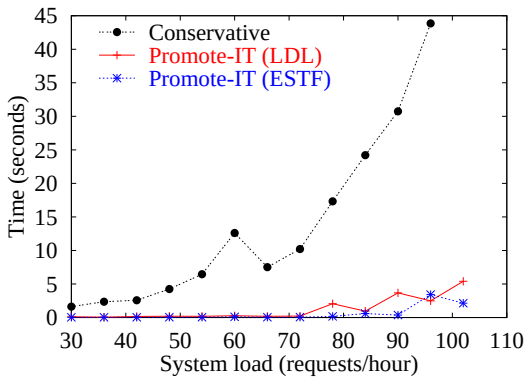
Periodic schedulers have a clear advantage over aperiodic schedulers in the computing time, because they just need to evaluate a couple of formulae to decide if a request is schedulable. However, this advantage is not visible to the end user, who notices only the response time and the confirmation time. When evaluating the performance of the optimal scheduler, we will show that the computing time becomes an important parameter when it influences the confirmation time.

We conclude that periodic scheduling is a bad technique for scheduling a jukebox, because even the FSBS scheduler—which is extremely simple—in many cases performs better than JEQS. The bad performance of JEQS is not a characteristic of this particular scheduler, but is intrinsic to any periodic jukebox scheduler. A periodic scheduler either needs to use the robot in a cyclic way, or take into account the worst-case time for robot contention in the execution time of the tasks. Therefore, when using a periodic scheduler, the best-case starting time for a request that does not produce a cache-hit is the maximum time needed to switch all the drives, even if the system load is very low and all drives are idle. In the same scenario, the starting time for Promote-IT is in most cases just the time to load the RSM in the drive and read the data of the first request unit. For FSBS it is the time needed to stage all the data of the request.

The response time of Promote-IT is also shorter than that of FSBS, as can be seen in Figure 1(a). FSBS stages the whole file before giving access to the user. Therefore, the response of FSBS has as lower limit the time to buffer the whole file, while the lower limit for Promote-IT is the time to buffer the first request unit.



(a) Mean Response Time.



(b) Mean confirmation time.

Figure 3: Early vs. conservative dispatching

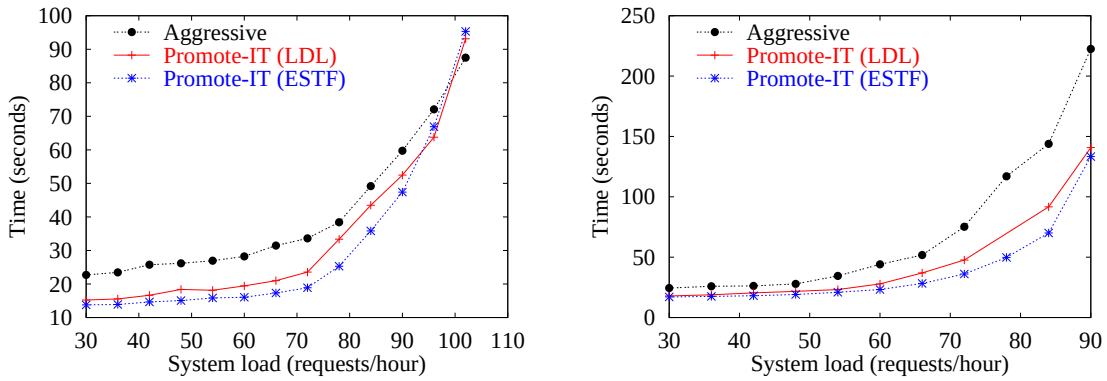
Moreover, Figure 2 shows that the difference in performance between Promote-IT and FSBS is even bigger when the data of a request is stored in multiple RSM. In this case, FSBS needs to perform multiple switches before giving access to the data, while in most cases Promote-IT only needs to perform one switch to read the data corresponding to the first request unit and the rest of the switches are performed at a later time, when the scheduler finds time for them. In this case the data in the jukebox consists of 30% long videos, 30% short videos, 30% music and 10% discrete data. The requests follow that pattern as well. The data of a request may be stored in multiple RSM.

Figure 3(a) compares the response time of Promote-IT and the extended conservative strategy (denoted as 'Conservative'). The main difference between LDL and the extended conservative strategy is the early dispatching of the tasks. As the system load increases, the difference in performance between Promote-IT and Conservative grows very fast. At the highest load level plotted, Conservative is unable to handle the load, because the waiting queue is too long. The test set is the same one described for comparing Promote-IT and FSBS with the requested data stored in multiple RSM.

The response time and confirmation time of LDL and ESTF are very similar when compared against the corresponding times of Conservative. Furthermore, when the system load is high, LDL performs slightly better than ESTF. This reinforces the idea that Back-to-Front is an interesting scheduling mechanism, when it is combined with early dispatching.

The confirmation time of Promote-IT is also lower than of Conservative (see Figure 3(b)). Conservative often fails to schedule incoming requests, because the starting time they should be assigned is too far into the future. Thus, the requests stay in the queue of unscheduled requests until the scheduler can incorporate them to the schedule.

The robot and drive utilization of Conservative is much less than that of LDL. When not using early dispatching, the resources are left idle, even if there are tasks in the schedule. Thus, when new requests arrive, their chances to be scheduled immediately are lower, even when the system load is low, because the scheduler has tasks scheduled for the future.



(a) Fast Jukebox (b) Slow Jukebox

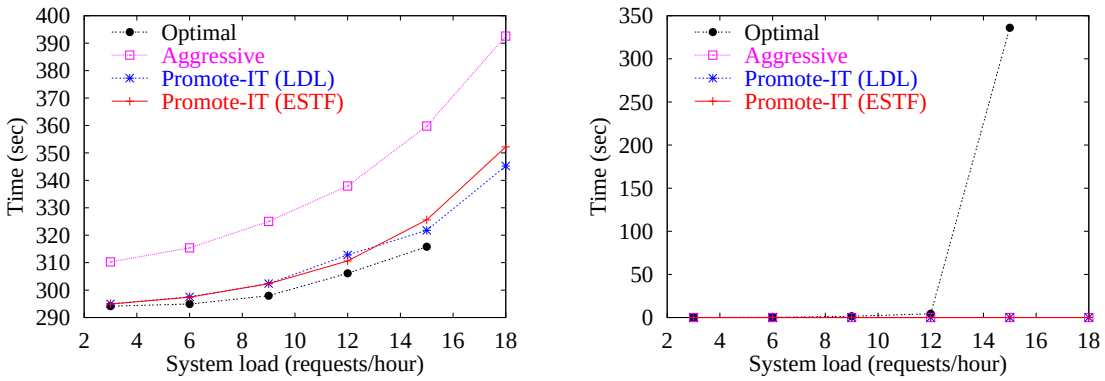
Figure 4: Uncoupled vs. Coupled load and unload. Mean Response Time

Figure 4 compares the response time of Promote-IT and the extended aggressive strategy (denoted as 'Aggressive'). The main difference between Aggressive and the ESTF strategy of Promote-IT is that Aggressive couples the load and unload into a single switch operation. This means that the RSM stay loaded in the drives until the drives are needed again. Therefore, Aggressive needs to perform first an unload before using a drive, even if the drive and the robot are idle before the request arrival.

When the system load is low and medium, Promote-IT provides shorter response times than Aggressive. However, when the system load is high and the robot is a clear bottleneck in the system, as in the case plotted in Figure 4(a), Aggressive has a better mean response time than Promote-IT. In this situation, the response time of Aggressive is similar to that of LDL, although Aggressive builds schedules Front-to-Back and LDL builds them Back-to-Front. However, Aggressive delays the last unload of a drive as much as possible, until the drive is needed again, which is the original goal of a Back-to-Front strategy. When the system load is low or medium, it is highly probable that at the time when a new request arrives there are idle resources. Therefore, delaying the unloads as much as Aggressive does affects the performance negatively. When the load is high it does not really matter, because there is no opportunity to unload the drives early anyhow. When the robot is not a strong bottleneck, as in the case plotted in Figure 4(b), Promote-IT provides shorter response times than Aggressive, even under high system loads. In this case unloading late is not beneficial: also ESTF performs better than LDL.

The request set used to compare Promote-IT and Aggressive is the same as the one used for FSBS and the extended conservative strategy. The 'Fast Jukebox' has the same configuration as previously described, in this configuration the robot is a clear bottleneck. The 'Slow Jukebox' has four DVD drives based on CLV technology with a transfer speed of 7.96 MBps and an maximum access time of 1.5 seconds.

Figure 5(a) shows that the response time provided by Promote-IT is near the optimal response time. Moreover, the difference in response time between Promote-IT and the optimal scheduler is smaller than the difference between Aggressive and Promote-IT. The plots



(a) Mean Response Time.

(b) Mean Computing Time.

Figure 5: Heuristic vs. Optimal

indicate that the difference in response time between Promote-IT and the optimal is larger as the system load increases. Therefore, we regret that we cannot run the optimal scheduler with higher loads.

The computing time of the optimal scheduler increases exponentially when the load of the system increases, while the computation time of the heuristic schedulers is nearly constant (see Figure 5(b)). The computing times of the optimal scheduler are so high that the scheduler cannot be used in an on-line system.

Additionally, we have observed that the optimal scheduler does not unload an RSM before all the data has been read in any of the runs we have performed. This is an important result in favor of the minimum switching model, on which Promote-IT is based, because even if the optimal scheduler has the possibility of performing intermediate switches, it does not do so.

The request set consists of 200 ASAP requests for long videos. The optimal scheduler does not deal with the cache administration. Therefore, each request corresponds to a different video and the cache is empty at the beginning of the runs. Thus, there are no cache-hits.

The jukebox only contains long videos, which were generated in the same way as those described in the comparison against JEQS. However, the data in the jukebox is stored in single-layered DVDs and the drives use CLV technology with a transfer speed of 6.45 MBps and a constant access time of 0.1 seconds.

When building the request sets we have to make a trade-off between keeping the number of request units per request low and having more than one request unit per RSM. The computational complexity of the optimal scheduler increases exponentially with the number of request units to schedule, so we should only split each file in a small number of request units. On the other hand, we want to give the optimal scheduler the possibility to switch an RSM without reading all the requested data from the RSM. Therefore, it is desirable to have more than one request unit per RSM. In the tests that we show here, we chose to

	FSBS	Extended Aggressive Strategy	Extended Conservative Strategy	Promote-IT	JEQS	Optimal
Flexibility: requests	++	++	++	++	−−	+
Flexibility: hardware	++	+	++	++	−	−
Response time	−−	+	−	++	−−	+++
Confirmation time	+	++	+	++	−	−−−
Computing time	++	++	++	++	+++	−−−
Deal with high load	+	++	−−	++	−−	−−−

Table 1: Summary of the performance comparison. The notation used is: excellent (+++), very good (++), good (+), bad (−), very bad (−−), and unusable (−−−).

chop the files in request units of 2.5 GB. Thus, the number of request units per request is between 1 and 4 and the number of RSM involved is 1 or 2.

Throughout this section we have shown that Promote-IT performs better than the other schedulers. However, the magnitude of the performance difference varies in each case. We put the differences in context and compare all the schedulers among each other.

We evaluate the capacity of the schedulers to deal with flexible requests and hardware. We also evaluate the schedulers regarding the response time, confirmation time, computing time and the capacity to deal with high load. Table 1 summarizes the evaluation. The classification we assigned to the schedulers in the last four categories is the result of observing their performance in multiple test setups. Although, the classification is quite subjective and difficult to quantify, we believe that it reflects correctly the average performance of the schedulers. Note that the original conservative and aggressive strategy and FSBS can only handle very limited types of jukeboxes and requests. The extensions we performed are discussed in [17].

5 Conclusions

Through Promote-IT we show that tertiary storage can be used effectively in systems with real-time requirements, for instance in a hierarchical multimedia archive. However, careful scheduling is needed in order to provide those guarantees, to use the resources efficiently, and to provide short response times to the users. A performance comparison of different schedulers, shows that Promote-IT performs better than the other heuristic schedulers, and additionally provides response-times near the optimum in cases where the optimal scheduler can be evaluated.

References

[1] J. Boulos and K. Ono. Continuous data management on tape-based tertiary storage systems. In *Proc. of the 5th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 290–301, Sept. 1998.

[2] D. W. Brubeck and L. A. Rowe. Hierarchical storage management in a distributed VoD system. *IEEE Multimedia*, 3(3):37–47, 1996.

[3] H. Cha, J. Lee, J. Oh, and R. Ha. Video server with tertiary storage. In *Proc. of the Eighteenth IEEE Symposium on Mass Storage Systems*, April 2001.

[4] S.-H. G. Chan and F. A. Tobagi. Designing hierarchical storage systems for interactive on-demand video services. In *Proc. of IEEE Multimedia Applications, Services and Technologies*, June 1999.

[5] A. L. Chervenak. *Tertiary Storage: An Evaluation of New Applications*. PhD thesis, Dept. of Comp. Science, University of California, Berkeley, December 1994.

[6] A. L. Chervenak. Challenges for tertiary storage in multimedia servers. *Parallel Computing*, 24(1):157–176, Jan. 1998.

[7] J. L. Cole and M. E. Jones. The IEEE storage system standards working group overview and status. In *Proc. of the 14th IEEE Symposium on Mass Storage Systems*. IEEE, September 1995.

[8] C. Federighi and L. A. Rowe. Distributed hierarchical storage manager for a video-on-demand system. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 185–197, February 1994.

[9] Fujitsu. Storage management overview. White Paper, January 1998.

[10] C. Georgiadis, P. Triantafillou, and C. Faloutsos. Fundamentals of scheduling and performance of video tape libraries. *Multimedia Tools and Applications*, 18(2):137–158, 2001.

[11] S. Ghandeharizadeh and C. Shahabi. On multimedia repositories, personal computers, and hierarchical storage systems. In *Proc. of th ACM Multimedia Conference*, 1994.

[12] L. Golubchik and R. K. Rajendran. A study on the use of tertiary storage in multimedia systems. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, March 1998.

[13] B. K. Hillyer and A. Silberschatz. Random I/O scheduling in online tertiary storage systems. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 195–204, June 1996.

[14] P. G. Jansen, F. T. Y. Hanssen, and M. E. Lijding. Scheduling of early quantum tasks. In *15th Euromicro Conf. on Real-Time Systems*, pages 203–210. IEEE Computer Society Press, Jul 2003.

[15] S.-W. Lau and J. C. S. Lui. Scheduling and replacement policies for a hierarchical multimedia storage server. In *Proc. of Multimedia Japan 96, International Symposium on Multimedia Systems*, March 1996.

[16] S.-W. Lau, J. C. S. Lui, and P. Wong. A cost-effective near-line storage server for multimedia system. In *Proc. of the 11th International Conference on Data Engineering*, pages 449–456, March 1995.

[17] M. E. Lijding. *Real-Time Scheduling of Tertiary Storage*. PhD thesis, CTIT Ph.D.-thesis Series 03-48, Univ. of Twente, May 2003.

[18] M. E. Lijding, P. G. Jansen, and S. J. Mullender. Implementing and evaluating jukebox schedulers using JukeTools. In *20th IEEE Symp. on Mass Storage Systems*, pages 92–96, San Diego, California, Apr 2003. IEEE Computer Society Press, Los Alamitos, California.

[19] C. Moon and H. Kang. Heuristic algorithms for I/O scheduling for efficient retrieval of large objects from tertiary storage. In *Proc. of the Australiasian Database Conference*, pages 145–152. IEEE, February 2001.

[20] S. More and A. Choudhary. Scheduling queries on tape-resident data. In *Proceeding of the European Conference on Parallel Computing*, 2000.

[21] H. Pang. Tertiary storage in multimedia systems: Staging or direct access? *ACM Multimedia Systems Journal*, 5(6):386–399, November 1997.

[22] S. Prabhakar, D. Agrawal, A. E. Abbadi, and A. Singh. Scheduling tertiary I/O in database applications. In *Proc. of the 8th International Workshop on Database and Expert Systems Applications*, pages 722–727, September 1997.

[23] D. Teaff, D. Watson, and B. Coyne. The architecture of the High Performance Storage System (HPSS). In *Proc. of the Fourth NASA GSFS Conference on Mass Storage Systems and Technologies*, 1995.

[24] P. Triantafillou and I. Georgiadis. Hierarchical scheduling algorithms for near-line tape libraries. In *Proc. of the 10th International Conference and Workshop on Database and Expert Systems Applications*, pages 50–54, 1999.

[25] P. Triantafillou and T. Papadakis. On-demand data elevation in hierarchical multimedia storage servers. In *Proc. of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 226–235, 1997.